DOCUMENT RESUME

ED 398 886 IR 018 066

AUTHOR Goldenson, Dennis
TITLE Why Teach Computer Programming? Some Evidence about Generalization and Transfer.
SPONS AGENCY National Science Foundation, Arlington, VA.
PUB DATE 96
CONTRACT MDR-8652015
NOTE 16p.; In: Call of the North, NECC '96. Proceedings of the Annual National Educational Computing Conference (17th, Minneapolis, Minnesota, June 11-13, 1996); see IR 018 057.
PUB TYPE Reports - Evaluative/Feasibility (142) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Academic Achievement; Authoring Aids (Programming); Computers; Computer Software; Educational Objectives; Generalization; *Grade 9; Instructional Effectiveness; Learning Processes; *Programming; Programming Languages; Secondary Education; *Skill Development; *Thinking Skills; *Transfer of Training

ABSTRACT

The assertion that "higher order" thinking skills can be improved by learning to program computers is not a new one. The idea endures even though the empirical evidence over the years has been mixed at best. In fact, there is no reason to expect that all programming courses will have identical, or even similar, effects. Such courses typically differ more by the languages in which they are taught than by anything else, and rarely do they explicitly address higher level instructional goals. To properly assess the extent of transfer, or any other learning, empirical measures must be criterion-referenced to specific curriculum objectives. This paper describes the results from three field studies. In two of them, ninth graders who learned structured programming methods using the "Karel the Robot" teaching language performed considerably better on a series of expository writing tasks than did students in the studies' control groups. In the third study, students who began their introductory programming methods course with Karel performed substantially better on difficult structured programming tasks using Pascal. (Author/SWC)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

Paper

Why Teach Computer Programming? Some Evidence About Generalization and Transfer*

Dennis Goldenson

Software Engineering Institute

Carnegie Mellon University

Pittsburgh, PA 15213-3890

412.268.8506

dg@sei.cmu.edu

ED 398 886

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☐ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

D. Ingham

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Key Words: **transfer, generalization, structured programming, higher order thinking skills, Karel**

Abstract

The assertion that "higher order" thinking skills can be improved by learning to program computers is not a new one. The idea endures even though the empirical evidence over the years has been mixed at best. But, there is no reason to expect that all programming courses will have identical, or even similar, effects. Such courses typically differ more by the languages in which they are taught than by anything else, and rarely do they explicitly address higher level instructional goals. To properly assess the extent of transfer, or any other learning, empirical measures must be criterion referenced to specific curriculum objectives.

This paper describes the results from three field experimental studies. In two of them, ninth graders who learned structured programming methods using the Karel the Robot teaching language performed considerably better on a series of expository writing tasks than did students in the studies' control groups. In the third study, students who began their introductory programming methods course with Karel performed substantially better on difficult structured programming tasks using Pascal.

Programming and Transfer

Why should students study computer programming? Aside from the pressing need to educate properly future programmers and software engineers, one often hears the argument that programming generates transferable problem solving and thinking skills. But is programming really the new Latin?[1, 35]

Undoubtedly it is not, if we are looking for evidence of widespread, incidental transfer to all possible areas of higher level cognitive processes[38, 34, 14, 33, 20, 16]. Transfer may be widely touted as a reason for teaching programming, but the assertion is based on mixed results empirically,[1] with the strongest claims often based only on anecdotal

[1]. Actually the positive effects shown in some studies are only of weak magnitude. The size of effects should not be confused with their statistical significance.[21, 37, 22]

* This material is based on work supported by the National Science Foundation under grant number MDR-8652015. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the foundation. The research was completed prior to the time that the author joined the Software Engineering Institute and should not be construed as an official position of the Software Engineering Institute.

Thanks to David Kaufer for his many insights about the writing process, and the original idea behind this study. Michele Matchett coauthored the CMU Writing Quality Rubric that appears in this paper's Appendix, graded the Minneapolis essays, and participated in many productive discussions about the paper's content.

IR018066

2 BEST COPY AVAILABLE

evidence. Clearly the case for transfer from programming has been overstated. Indeed introductory programming courses often have been characterized by disturbingly high failure and drop rates. We seem to have enough trouble teaching the narrow subject matter[16, 12], never mind transfer of generic thinking skills. Yet the insight still remains compelling to many educators.

Recent empirical work does show more evidence of transfer than some critics have conceded[29, 26]. However a great deal more remains to be learned about the conditions where transfer reasonably can be expected to occur[7, 10, 32, 31].

Programming deals, or at least can deal, in structure and abstraction. Yet it does so in an uncommonly concrete and tangible manner: programs are realized in a physical machine, that students can touch and on which they can see immediate results of their logic.[2] Unfortunately, though, programming languages are only tools, and certainly do not assure that curricula and teachers will focus explicitly on generalizable problem solving skills[11, 36, 19, 19, 4, 14, 33, 16].

Many introductory programming courses limit their attention to writing short, syntactically correct, but contrived programs. Serious attention to topics such as procedural abstraction are often postponed to more "advanced" courses. Explicit consideration of debugging strategies tends to be neglected altogether[4]. There is little if any reason to expect such courses to teach programming very well, much less transfer to other domains[12].

So What Ought We be Teaching Anyway?

Properly done, introductory programming courses can foster at least two important sets of thinking skills: disciplined attention to details; and high level abstraction/planning/organizational/design skills. Though both sets of skills characterize programming proficiency, the former usually gets emphasized by default. Students write short programs graded on the basis of whether they "work," because getting them to work requires considerable attention to error prone syntax and tool invocation details. The latter barely get spoken to, since the programs novice students write tend to be so short and lacking in abstract complexity. Hence we reward tenacity rather than conceptual clarity in design and implementation.

Methodological Confounds

Many transfer studies suffer from an all too common fault: namely actuarial description is not the same as explanation[9, 13]. Indeed, one ought to expect minimal effects when students who have completed many different, widely divergent, programming courses are lumped into the same predictor categories[21]. Rather, we need knowledge about the specifics of curricula to design appropriate criterion referenced indicators of effect.

Moreover it may be impossible to do the definitive experimental study in field conditions. Internal validity is bound to be compromised in the classic trade-off for external validity. Too often we have small Ns, and less than ideal design and

Thanks also to Bill Hefley, Mark Shermis, Bing Jyun Wang, Mei-Hung Chiu, and the MacGNOME group at Carnegie Mellon. Special thanks to the teachers who have participated in the various stages of this research: JoAnn Avery, Don Baker, Jim Bowlby, Jane Bruemmer, Larry Faulk, Laura Gardner, Joyce Hotchkiss, Bruce McClellan, Mary Northcutt, John Sutula, and Jim Turpin.

† Software Engineering Institute, Pittsburgh, PA 15213; 412/268-8506; dg@sei.cmu.edu.

2. At least since Socrates, educators have stressed the importance of active involvement by students in the learning process. Similar ideas underlie notions of inquiry, case studies and "constructionist" learning.

experimental subject selection criteria. So confidence in results must come through internal consistency and replication.[3]

The three field experiments reported here were done as spin-offs of a larger project that focused on the development and evaluation of structure editor based programming environments[6, 11, 12]. Most resources in the larger project were devoted to the development of the software, and evaluation of its value in the learning by novice programmers of software engineering skills and knowledge.

The external validity is quite defensible in all three studies. Yet the Ns are small and the control conditions are imperfect. Still, several individual relationships are statistically significant. Moreover the results across the various relationships are quite consistent, which is highly unlikely by chance, especially with small numbers of cases[28].

Transfer or Associative Learning?

Clearly it is difficult to establish conditions that induce extensive incidental learning without explicit instructional intervention. Transfer in that sense is undoubtedly rare. Only the most outstanding students are likely to apply what they have learned in one domain to another in a truly novel manner. But it is more than enough to demonstrate associative learning, where higher level abstractions are explicitly taught and connected across traditionally separate subject matter boundaries. It is difficult enough to establish direct learning effects in educational research. I am more than happy to show evidence of associative learning across the curriculum.

Transfer from Programming to Writing

On the surface, writing is an unlikely candidate for "far transfer" from programming. Yet both fields share a common concern for planning, organization, structure and design. However properly presented programming methods courses can get to the essentials of planning and design much more quickly and easily than can writing courses. Introductory programming language and problem semantics are much simpler than those found in a rich (and vague and ambiguous) spoken language. A common problem for writing teachers is that students must spend a great deal of time learning and summarizing the basic facts of a topic before they are ready for any serious attempts at synthesis and analysis. Early emphasis is by necessity on the concrete before the abstract[15].

Karel the Robot and the Karel GENIE

Richard Pattis' *Karel the Robot* [27] teaching language has extremely simple semantics, allowing it to be taught in a very short period of time, with early and sustained emphasis on the procedural abstraction of complex tasks. The GENIE syntax directed structure editing environment[4] simplifies programming in Karel even further, while also providing a powerful set of online design and testing tools.

The Karel Language

Named for Karel Capek, the Czechoslovakian playwright who wrote *R.U.R.* (Rossum's Universal Robots), Karel indeed is a pleasant and "gentle introduction to the art of programming." Equally important, it is a serious high level language that focuses

[3]. The importance of replication is fundamental to science. Unfortunately the notion remains foreign to too many scientists who crave definitive proof.

[4]. The Karel GENIE is distributed by the Chariot Software Group. For further information or to place an order (catalog # 570-0801), contact them at 800/CHARIOT. A Pascal GENIE is also available through Chariot.

immediately on the essentials of programming method and problem solving. Even students who initially are reticent about programming find themselves learning a great deal before they realize what has happened.

Karel is an imaginary robot who moves through a Cartesian coordinate quarter-plane of "streets" and "avenues." It is easy for students (and teachers) to anthropomorphize problems in Karel's "worlds." But the problems are by no means trivial intellectually.

The language requires very little attention to extraneous detail. It is a sparse language, with only *five* primitives, almost no concrete syntax (none with a structure editor), "no" variables, and *no* subtle input/output details. Karel is easy to learn, even for middle school-aged children. Thus novices can attempt some rather non-trivial problems after a very short period of study.

An excellent introduction to programming methodology, Karel introduces key structured programming concepts, including the major logic control structures of many other "more serious" high level languages. Yet, because of the absence of distracting details, Karel also is actually more amenable than "real" languages to early serious study of control abstraction.

Of particular value in an expository writing context is Karel's emphasis on modular programming and procedural abstraction. This is accomplished through user definition of new instructions, which are functionally equivalent to Pascal parameterless procedures.

The language was designed to enable teachers to emphasize top-down, modular design skills that would be transferable to programming in other algorithmic languages such as Pascal. The anecdotal evidence from those who have taught, and learned, with Karel is that it can perform that role admirably well. Karel's combination of new instructions and simple problem semantics does seem to help students to tackle conceptually complex search and traversal problems in an organized, intellectually manageable manner.

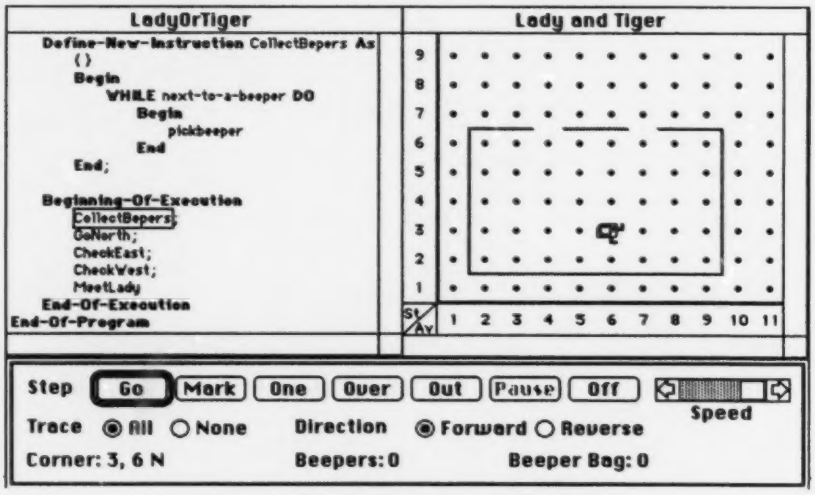

Figure 1. Karel's Run Time System:
Tracing Karel's Movement Through One of His Worlds

BEST COPY AVAILABL

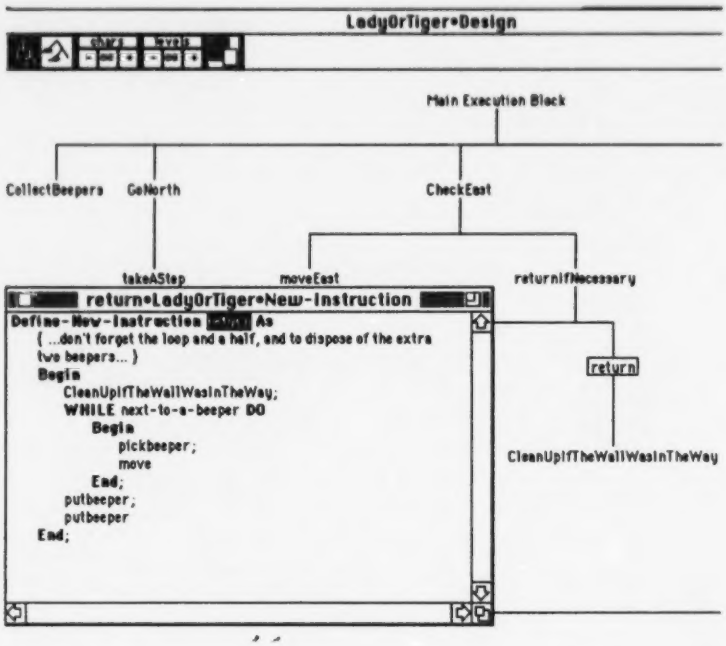

Figure 2. Karel's Design View & a Scope View

The Programming Environment

As shown in Figure 1, the Karel GENIE is a "seamless" environment that integrates syntax free structure editing with high level "CASE[5] like" design tools and powerful run time testing and debugging tools.[6] The basic structure editing functionality and run time system make it even easier to cover Karel and programming methods comprehensively with young (and not so young) children in a very short time period.

The structure editor based design tools may be even more important[30, 6]. It is possible to display concurrently different aspects of a program in alternative "views" that are *not* limited to the program's concrete syntax. For example the "design" view depicted in Figure 2 allows the programmer to focus attention in a hierarchical, graphical manner on the program's high level procedural call structure. "Scope" views, such as the one also shown in Figure 2 may be used to attend to lower level implementation details. Since program structure is maintained in an underlying structure database, changes made in any one view are reflected elsewhere immediately. In the case of the design view, one can use it to understand the structure of an already existing program, and/or to map program code from the graphical representation.

Results

As already indicated, initial analyses have found statistically (and substantively) significant results, showing transfer from Karel (or at least associative learning) in two separate domains thus far: "far transfer" to expository writing; and "near transfer" to

5. "CASE" is an acronym for Computer Assisted Software Engineering.

6. GENIE also adds recursion and comments to the Karel language.

BEST COPY AVAILABLE

programming performance in a standard structured language (Pascal). The results of three separate studies are reported here.

Expository Writing -1

In this study a group of Minneapolis ninth graders learned Karel the Robot in the context of a team taught, interdisciplinary science course. Another group covered a unit on the mechanics of word processing taught by the same teacher during the same time period.[7] Students in both groups completed the same major writing assignment: a research design for a year long project in field biology, chemistry and/or ecology.

The two groups were chosen on the basis of administrative convenience (periods 4 & 5). However we are aware of no known prior selection effects. Students in both classes had similar overall course schedules and extra curricular activities and similar grade point averages.

We emphasized several parallels between good programming and good writing, both in structuring our Karel curriculum unit and assessing writing quality. The writing samples were blind graded, following the grading rubric that is reproduced as an appendix to this paper. Following similar assessment procedures with high inter coder reliability used in Columbus, Ohio, California, and elsewhere[8, 3, 25], the rubric focuses on structure and design criteria. As seen in the appendix, we drew explicit parallels from our own computer science program grading rubrics, informed by the theoretical framework implicit in previous writing assessment rubrics.

The student science project research designs were completed about ten weeks after the Karel unit. Unlike the short essays so prevalent in English classes, these writing samples tend to be longer (up to ten pages) and are based on topics that the students knew well.

All of the students received the same exhortations to structure their essays in a well organized fashion, but the computer science teacher did draw specific analogies with programming methods for the Karel group. Hence any effects can be better characterized as evidence of associative learning as opposed to true incidental transfer.

As seen in Tables 1 through 4, the Karel group did indeed perform considerably better on blind graded writing samples. Interestingly enough, on first reading, the grader (a writing teacher) was struck by the poor grammar and spelling in both groups' essays. However, using the CMU rubric to assess organizational structure, the papers basically fell into two piles.

Table 1. Median Writing Quality Scores*

	Karel	Control
Expressed as:		
Raw Score†	20	16
% of Total Possible	77	56
X Component Score	3.67	2.67
N =	16	16

7. The Karel unit lasted about four weeks, after which the experimental students also were introduced to the word processing topics covered by the control group. The teacher's impression was that the Karel students in fact did equally well on the word processing tasks. They seemed to enjoy the work with Karel, which also introduced them to common Macintosh editing conventions.

* Individual student scores are simple summed composites of all six grading criteria summarized in the appendix.

† p = .05 according to Mann-Whitney U-Test Criteria

Table 2. Categorized Writing Quality Scores*

	Karel	Control
highest (23–24)	44%	31%
high (18–22)	44	6
low (16–17)	12	25
lowest (13–15)	0 100%	38 100%
N =	16	16

* Individual student scores are simple summed composites of all six grading criteria summarized in the appendix.

p = .01 according to Chi^2 Criteria

Table 3. Repetition Rubric Criterion Only*

	Karel	Control
highest (4)	75%	31%
high (3)	19	69
low (2)	6 100%	0 100%
N =	16	16

* p = .01 according to Chi^2 Criteria

Table 4. Hierarchy Rubric Criterion Only*

	Karel	Control
highest (4)	38%	38%
high (3)	56	19
low (2)	6 100%	44 100%
N =	16	16

* p = .02 according to Chi^2 Criteria

Tables 1 and 2 report the summary results using a simple summed index based on all six grading rubric criteria summarized in this paper's appendix. Individual items are notoriously unreliable. But it is worth noting that the Karel group performed better on all six component indicators. Even with the small number of cases in the experiment,

the repetition and hierarchy differences reported in Tables 3 and 4 are statistically significant.[8]

Expository Writing -2

Conceived as a pilot study, this experiment was conducted with ninth graders in a Kansas City suburb. In it we were less concerned with experimental design niceties than with demonstrating to ourselves that the idea had merit. We were aware of the fact that Karel has been used in middle school computer science curricula. But we also wanted to satisfy ourselves that it is possible to teach the language in a short time period to less technically oriented students.[9] The results were encouraging in their own right.

Two groups of students were drawn randomly from the same ninth grade language arts class. One group left class early in the semester for a three week unit on Karel taught by the school's lead computer science teacher.[10] The other group did Karel at semester's end. The entire class completed three writing assignments in the interim.

Grading of the three assignments was done by the writing teacher who participated in the study. Although the grading was not done blind, the teacher did pool all of the assignments into one group rather than grading them separately, and reported that he indeed was unaware of the students' group membership while grading. The grading was based on the participating teacher's explicit organizational criteria, which are summarized in a note to Table 5.

Table 5. Pilot Study Writing Performance Median Scores*

	Karel First	Control	p = †
Three Paragraph Work			
Raw Score	97	83	.04
Residual	+9	-5	.44
Three Paragraph Paper			
Raw Score	69	21	.01
Residual	+10	-20	.04
Reading Test			
Raw Score	75	59	.02
Residual	+2	-3	.19
Karel Grade	84	80	.04
N =	13	13	—

* Grades on the three writing assignments were based on the following criteria, and constrained to a zero to 100 point scale by the participating teacher: topic sentence=1; explicit conclusion=1; supporting details=1 each; coherence (transitional device and direct references to other related information=3; one of the above=2; none used, can follow=1; can't be followed=0). The Karel grade

8. Of course the probability that all six would consistently favor the Karel group is highly unlikely by chance alone, especially given the small N. Remember the multiplication rule[28].

9. We also have begun work with third graders, who seem quite capable of learning Karel.

10. She also taught Karel to the writing teacher.

was a summary performance grade assigned by the computer science teacher who taught the Karel unit.

† According to Chi^2 Criteria

As can be seen in the table, the group who completed the Karel unit first did better on all three writing assignments. But there was a confound: they also did somewhat better on Karel. Perhaps they were better motivated earlier in the academic term and/or the Karel work itself was a good motivator, such that they paid better attention to their subsequent writing tasks. In spite of the random assignment, possibly there was indeed a selection effect with the small number of cases.

Hence I also calculated residual scores, based on the difference between the students' actual scores on each of the three writing assignments and their predicted scores given the positive relationship between performance on Karel and the writing quality measures. In the residuals analysis the Karel first group still did better on all three assignments. Given the small number of cases only one relationship was statistically significant but the characteristic differences were present in all three instances.[11]

Of course one can attribute the results to explicit instruction and resultant associative learning rather than to "true" incidental transfer. As already noted, the writing teacher did become somewhat familiar with Karel during the experimental period. The students also may have realized that their teachers thought programming experience could impact on their writing. But, unlike in the Minneapolis study, explicit parallels between Karel and writing were *not* emphasized in the writing class.

Near Transfer to Programming in Pascal

A major purpose of using Karel in an introductory programming methods course is that it facilitates very early attention to non-trivial issues of procedural (and control) abstraction. Such abstractions presumably make for a simpler transition to Pascal, where students will expect to use modular programming as a matter of course.

In this last study Karel was taught as an introduction to work in Pascal by ninth graders in a Pittsburgh area private school. One group studied Karel for about six weeks in total, with heavy emphasis on procedural abstraction and explicit consideration of relatively complex control abstraction as well. This group thus spent comparably less time on Pascal as a result.

The other group also did some Karel. The participating teacher felt strongly that it was important to use Karel enough to familiarize the students with the GENIE software environmental semantics. However these students were quickly weaned from Karel, having worked with no procedurally complex Karel programs.

The two groups of students were chosen on the basis of administrative convenience from the school's introduction to computing course. However, based both on the teacher's judgment in a small class where he knew the students well and a review of school records, there were no known selection effect differences between the groups.

11. Once again note that consistency across relationships is much less likely to occur by chance alone than is a single significant relationship.

Table 6. Near Transfer from Karel to Pascal

	Karel	Control
Game Program		
median	85	85
first quartile	80	56
% failing grade*	0	25
Course Grade		
median	85	83
first quartile	79	56
% failing grade*	0	25
N =	9	12

* p = .05 according to Chi^2 Criteria

As shown in Table 6, the Karel group in fact did better on both the "capstone" final program assignment and on final course grades. The programs were graded for both procedural abstraction and functionality.[12] The Karel group's programs were both better structured procedurally, and tended to include more complex functionality and better looking graphical user interfaces.

Again the small number of cases make it unlikely that any one relationship is statistically significant. But the characteristic group differences remain. Moreover there were in fact significantly fewer failures in the Karel group.

Implications for Teaching Writing

Clearly teaching students to be good structured programmers will not obviate the need for teaching writing! As seen in the Minneapolis study, the Karel group's written essays showed room for considerable improvement in attention to spelling and similar "low level" syntax details. Of course such is not surprising; the Karel programming lessons purposefully ignored such issues. But what is impressive is that a short introduction to programming methods apparently did serve to help the same students better organize their essays for structure and readability.

Moreover the transfer seems to be *asymmetrical*.[13] That is, it seems to be easier to teach well structured programming than writing. Since the semantics of both the language and problem (with Karel) domains are simpler than in writing essays, there is far less detailed information to summarize before the students are asked to analyze and synthesize.

Of course we should continue to build (computer based) tools to teach writing skills[23, 23, 17, 2]. After all, computers are well suited for supporting data hiding, revision, hierarchical organizational and related tools. But schools also must do much more to encourage making connections across the curriculum.

---

12. The assignments were not graded blind. However the teacher did pool them into one group. Since so much time had passed following the Karel unit, he claimed to be unaware of group membership at the time of grading.

13. Of course none of the data directly support this assertion about asymmetrical transfer. But it seems both a very plausible conjecture and worthy of further research[5].

Conclusions

In the three studies reported here I have been able to demonstrate some intriguing transfer effects attributable to programming in the Karel the Robot teaching language. Of course such effects require proper attention in the classroom to issues of structure, organization, planning and design. Although this particular language does facilitate such attention, even it does not insure that teachers will focus on programming method and problem solving as opposed to language constructs alone.

All too often programming courses have been notoriously badly done. The response in many secondary schools has been to drop programming altogether, in favor of computer literacy courses organized around spreadsheets, word processors, databases and similar standard software applications. Yet some of us continue to believe that appropriate programming education is (and will become increasingly) essential for both pre-professional and general education[1, 35]. I hope that these results will contribute to a continuing dialogue and effort to improve programming education, with serious attention to the possibilities for computing across the curriculum.

References

[1] Bonar, J., "Everyone Will Be a Programmer," *Technology and Learning*, July 1987.

[2] Britton, B. K., and S. M. Glynn (eds), *Computer Writing Environments: Theory, Research, and Design*, Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1989.

[3] California Association of Teachers of English, "California Essay Scale," in *A Guide for Evaluating Student Composition*, Urbana, Illinois: National Council of Teachers of English, 1965.

[4] Carver, S., "Learning and Transfer of Debugging Skills: Applying Task Analysis to Curriculum Design and Assessment," in R. Mayer (ed), *Teaching and Learning Computer Programming: Multiple Research Perspectives*, Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1988.

[5] Carver, S., and M. Walker, "In Search of General Planning Skills," *Proceedings of NECC '89, National Educational Computing Conference*, Boston, June 1989.

[6] Chandhok, R., D. Garlan, D. Goldenson, P. Miller, and M. Tucker, "Programming Environments based on Structure Editing: The GNOME approach," *Proceedings of the 1985 National Computer Conference*, Chicago, 1985.

[7] Chatel, S., F. Detienne, and I. Borne, "Transfer among Programming Languages: An Assessment of Various Indicators," paper presented at the Fifth Workshop of the Psychology of Programming Interest Group, PPIG5, Paris, December 1992.

[8] Columbus Public Schools, *The Columbus Rubric for Writing*, Columbus, Ohio: Columbus Public Schools, 1987.

[9] Converse, P., "Attitudes and Non-Attitudes: Continuation of a Dialogue," in E. Tufte (ed), *The Quantitative Analysis of Social Problems*, North Reading, Massachusetts: Addison-Wesley, 1970.

[10] Fix, V., and S. Wiedenbeck, "Designing a Tool for Learning Ada Using Empirical Studies," Proceedings of *The Fifth Workshop of the Psychology of Programming Interest Group (PPIG5)*, Paris, December 1992.

[11] Goldenson, D., and B. J. Wang, "Use of Structure Editing Tools by Novice Programmers," in J. Koenemann-Belliveau, T. G. Mohr, and S. P. Robertson (eds), *Empirical Studies of Programmers: Fourth Workshop*, Norwood, NJ: Ablex Publishing Corporation, 1991.

[12] Goldenson, D., "Learning to Program with Structure Editing: An Update and Some Replications," *Proceedings of NECC '90, National Educational Computing Conference*, Nashville, June 1990.

[13] Hovland, C., "Reconciling Conflicting Results Derived from Experimental and Survey Studies of Attitude Change," *The American Psychologist*, 14 (1959).

[14] Johanson, R., "Computers, Cognition, and Curriculum: Retrospect and Prospect," *Journal of Educational Computing Research*, 4 (1988).

[15] Kaufer, D., C. Geisler and C. Neuwirth, *Arguing from Sources: Exploring Issues Through Reading and Writing*, New York: Harcourt-Brace, 1989.

[16] Kurland, D., R. Pea, C. Clement and R. Mawby, "A Study of the Development of Programming Ability and Thinking Skills in High School Students," in E. Soloway and J. Spohrer (eds), *Studying the Novice Programmer*, Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1989.

[17] Kurland, M., and A. Bardige, "Language: The Next Generation of Writing Tools," *Proceedings of NECC '89*, National Educational Computing Conference, Boston, June 1989.

[18] Linn, M., and M. Clancy, "Can Experts' Explanations Help Students Develop Program Design Skills?" *International Journal of Man-Machine Studies*, 37 (1992).

[19] Linn, M., and J. Dahbey, "Cognitive Consequences of Programming Instruction: Instruction, Access, and Ability," *Educational Psychologist*, 20 (1986).

[20] Mayer, R., J. Dyck and W. Vilberg, "Learning to Program and Learning to Think: What's the Connection?," *Communications of the ACM*, 29 (July 1986).

[21] McCoy, L., and N. Dodl, "Computer Programming Experience and Mathematical Problem Solving," *Journal of Research on Computing in Education*, (Fall 1989).

[22] Mohamed, M., *The Effects of Learning LOGO Computer Language Upon the Higher Cognitive Processes and The Analytic/Global Cognitive Styles of Elementary School Students*, unpublished Ph.D. thesis, University of Pittsburgh, 1985.

[23] Neuwirth, C., R. Chandhok, D. Kaufer, P. Erion, J. Morris, and D. Miller, "*Flexible Diff-ing in a Collaborative Writing System*," *Proceedings of CSCW '92 Sharing Perspectives*, Toronto, November 1992.

[24] Neuwirth, C., D. Kaufer, R. Chandhok, and J. Morris, "Issues in the Design of Computer Support for Co-authoring and Commenting," *Proceedings of CSCW '90 Conference on Collaborative Work*, Los Angeles, October 1990.

[25] Odell, L., "Defining and Assessing Competence in Writing," in C. Cooper (ed), *The Nature and Measurement of Competency in English*, Urbana, Illinois: National Council of Teachers of English, 1981.

[26] Palumbo, D. B., "Programming Language/Problem-Solving Literature: A Review of Relevant Issues," *Review of Educational Research*, 60 (Spring 1990).

[27] Pattis, R., *Karel the Robot: A Gentle Introduction to the Art of Programming*, New York: John Wiley & Sons, 1981.

[28] Payne, J., "Fishing Expedition Probability: The Statistics of *Post Hoc* Hypothesizing," *Polity*, 7(Fall 1974).

[29] Pennington, N., and R. Nicolich, "Transfer of Training Between Programming Subtasks: Is Knowledge Really Use Specific?" in J. Koenemann-Belliveau, T. G. Moher and S. P. Robertson (eds), *Empirical Studies of Programmers: Fourth Workshop*, Norwood, New Jersey: Ablex Publishing Corporation, 1991.

BEST COPY AVAILABLE

[30] Roberts, J., J. Pane, M. Stehlik and J. Carrasquel, "The Design View: A Design Oriented High Level Visual Programming Environment," *Proceedings of the IEEE 1988 Workshop on Visual Language*, Pittsburgh, October 1988.

[31] Scholtz, J., "Transfer by Experienced Programmers: A Longitudinal Study," *Proceedings of The Fifth Workshop of the Psychology of Programming Interest Group (PPIG5)*, Paris, December 1992.

[32] Scholtz, J., and S. Wiedenbeck, "Learning New Programming Languages: An Analysis of the Process and Problems Encountered," *Behaviour & Information Technology*, 2 (1992).

[33] Seidman, R., "New Directions in Educational Computing Research," in R. Mayer (ed), *Teaching and Learning Computer Programming: Multiple Research Perspectives*, Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1988.

[34] Singley, M. K., and J. R. Anderson, *The Transfer of Cognitive Skill*, Cambridge, Mass.: Harvard University Press, 1989.

[35] Soloway, E., "Why Kids Should Learn to Program," *Proceedings of the 6th Canadian AI Conference, Montreal*, 1986.

[36] Soloway, E., J. Spohrer and D. Littman, "*E UNUM PLURIBUS*: Generating Alternative Designs," in R. Mayer (ed), *Teaching and Learning Computer Programming: Multiple Research Perspectives*, Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1988.

[37] Tu, J., and J. Johnson, "Can Computer Programming Improve Problem-Solving Ability?" *ACM SIGCSE Bulletin*, 22 (June 1990).

[38] Wu, Q., and J. R. Anderson, "Knowledge Transfer among Programming Languages," *Proceedings of The Fifth Workshop of the Psychology of Programming Interest Group (PPIG5)*, Paris, December 1992.

Appendix: The CMU Writing Quality Rubric

Organization and Composition

Hierarchy (top-down design)

4 The central idea is stated very clearly and examples and details are used as support. The subordinate ideas/examples/details are stated separately in clearly connected hierarchical fashion. Good transitional words and phrases as well as sectional headers delineate the sections.

3 While the central idea is stated, subordinate ideas, examples and details are not always well developed to support and develop the main argument of the paper. The use of some transitional devices helps indicate the delineation of sections.

2 The central idea is vague with no development or supporting details/examples. Few, if any, transitional devices are used.

1 There is no central idea. There are also no examples or details and the author seems to have used "spaghetti logic" when writing this paper.

Sectioning (modularity)

4 Ideas/information are logically developed and expressed in a separate cohesive fashion (e.g. sections) instead of being interwoven with other ideas.

3 The ideas/information are presented in a logical manner, but the author may stray from the topic and occasionally interweave ideas or information.

2 The ideas/information are often interwoven which makes it difficult to determine how the author has sectioned the information.

1 There is no indication as to how the author has sectioned the ideas/information.

Distinction Between Detail and Abstraction (granularity)

4 Major ideas are well-emphasized by keeping details and examples distinct (perhaps but not necessarily in separate sections) from those ideas.

3 Although major ideas tend to be emphasized, examples and details are not always kept distinct.

2 The major ideas are often unclear because details and examples are intermingled with those ideas.

1 Details and examples are generally indistinguishable from major ideas.

Repetition (repetitive code)

4 The authors repeat information only when the repetition clarifies their arguments. When doing so, the authors use effective paraphrasing.

3 Although the author repeats information when it is necessary, he/she also occasionally repeats information when it is not necessary. The author does, however, usually use effective paraphrasing.

2 The author tends to be unnecessarily redundant and does not use paraphrasing effectively.

1 The paraphrasing and repetition of ideas/information suggest that little thought was given to the topic.

Information

General Data Definitions (data structure/definitions)

4 General classes/types of information are defined explicitly perhaps in a single location with any structural relationships among those classes stated explicitly.

3 While definitions and/or relationships are explicitly stated, structural relationships are sometimes omitted.

2 While definitions and/or relationships are stated, they are vague and/or incomplete.

1 General classes and structural relationships are rarely if ever defined.

Specific Instance Declarations (data declaration/instantiation)

4 Specific instances of information are explicitly described in the context where they are used or by explicit reference to another context if that information has been used there.

3 Specific instances of information are often defined or referenced and these definitions and references are usually in context.

2 Specific instances of information are sometimes described or referenced but usually not in context.

1 Rarely if ever is explicit information described or referenced.

Grading Score Sheet

____ Top/down design (hierarchy)
____ Modularity (sectioning)
____ Granularity (distinction between detail and abstraction)
____ Repetitive Code (repetition)
____ Data structure/definitions (general data definitions)
____ Data declaration/instantiation (specific instance declaration)
____ Total Score